Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 131 (2005) 39–49

www.elsevier.com/locate/entcs

On the Transformation of SystemC to AsmL Using Abstract Interpretation

Ali Habibi[1]

Electrical and Computer Engineering
Concordia University
Montreal, Canada

Sofiene Tahar[2]

Electrical and Computer Engineering
Concordia University
Montreal, Canada

Abstract

SystemC is among a group of system level design languages proposed to raise the abstraction level for embedded system design and verification. A straight and sound verification by model checking or theorem proving of SystemC designs is, however, infeasible given the object-oriented nature of this library and the complexity of its simulation environment. We illustrated, in a previous work, the feasibility and success of performing model checking and assertions monitors generation of SystemC using a variant of Abstract State Machines (ASM) languages (AsmL). In this paper, we establish the soundness of our approach by proving the correctness of the transformation from SystemC to AsmL.

Keywords: SystemC, Formal Verification, Abstract Interpretation.

1 Introduction

SystemC [11] is an object-oriented system level language for embedded systems design and verification. It is expected to make a stronger effect in the area of architecture, co-design and integration of hardware and software. The

[1] Email: habibi@ece.concordia.ca
[2] Email: tahar@ece.concordia.ca

doi:10.1016/j.entcs.2005.01.021

SystemC library is composed from a set of classes and a simulation kernel extending C++ to enable the modeling of complex systems at a higher level of abstraction than state-of-the-art HDL (Hardware Description Languages). However, except for small models, the verification of SystemC designs is a serious bottleneck in the system design flow. Direct model checking of SystemC designs is not feasible due to the complexity of the SystemC library and its simulator. To solve this problem, we proposed in [4] to translate SystemC models to an intermediate representation in AsmL [8] more suitable for formal verification. This approach reduced radically the complexity of the design at the point that we were able to verify a complex PCI architecture using the SMV model checker [12].

In this paper, we provide a formalization of the SystemC and AsmL semantics in fixpoint based on the OO general case given in [7]. Then, we prove that, for every SystemC program, there exists an AsmL program preserving the same properties, w.r.t. an observation function α. The basic concept of this proof of soundness is based on the systematic design of program transformation frameworks defined in [2]. Such a result will enable using a variety of formal tools (for e.g., SMV for model checking [12]) or to use AsmL tool (Asmlt) to generate a finite state machine of the design.

Related work to ours concerns in particular, defining the formal semantics of SystemC and AsmL. For instance, several approaches have been used to write the SystemC semantics (e.g., using ASM is [9]). Denotational semantics [10] is found to be most effective since objects can be expressed as fixpoints on suitable domains. Salem in [13] proposed a denotational semantics for SystemC. However, the proposal in [13] was very shallow missing to relate the semantics of the whole SystemC program to the semantics of its classes. Therefore, in order to construct a transformation relation between SystemC and AsmL and to prove its soundness, we define, in this paper, our own SystemC denotational semantics.

Regarding, the program transformation, the work of Patrick and Radhia Cousot in [2] is the essence for any program transformation using abstract interpretation. The tactical choice of using semantics to link the subject program to the transformed program is very smart in the sense that it enables proving the soundness proof of the transformation, related to an observational semantics. A projection of that generic approach, described in Section 3.9 of [2] on a SystemC subject program and an AsmL transformed program can be used to perform the soundness of a transformation and also to construct it. In both cases, we need to define the syntax, semantics and observation functions for both AsmL and SystemC.

The rest of this paper is organized as follows: Section 2 and Section 3

present, respectively, the SystemC and AsmL semantics in fixpoint. Section 4 contains the proof of the existence and soundness of the SystemC to AsmL transformation. Finally, Section 5 concludes the paper.

2 SystemC Fixpoint Semantics

2.1 Syntactical Domains

SystemC have a large number of syntactical domains. However, they are all based on the single `SC_Module` domain. Hence, the minimum representation for a general SystemC program is as a set of modules.

Definition 2.1 (SystemC Module: `SC_Module`)
A SystemC Module is a set ⟨`DMem`, `Ports`, `Chan`, `Mth`, `SC_Ctr`⟩, where `DMem` is a set of the module data members, `Ports` is a set of ports, `Chan` a set of SystemC Chan, `Mth` is a set of methods (functions) definition and `SC_Ctr` the module constructor.

Definition 2.2 (SystemC Port: `SC_Port`)
A SystemC Port is a set ⟨`IF`, `N`, `SC_In`, `SC_Out`, `SC_InOut`⟩, where `IF` is a set of the virtual methods declarations, `N` is the number of interfaces that may be connected to the port, `SC_In` is an input port (provides only a `Read` method), `SC_Out` is an output port (provides only a `Write` method) and `SC_InOut` is an input/output port (provides `Read` and `Write` functions).

In contrast to default class constructors for OO languages, the SystemC module constructor `SC_Ctr` contains the information about the processes and threads that will be executed during simulation.

Definition 2.3 (SystemC Constructor: `SC_Ctr`)
A SystemC Constructor is a set ⟨`Name`, `Init`, `SC_Pr`, `SC_SSt`⟩, where `Name` is a string specifying the module name, `Init` is a default class constructor, `SC_Pr` a set of processes and `SC_SSt` is a set of sensitivity statements (to set the process sensitivity list `SC_SL`).

Definition 2.4 (SystemC Process: `SC_Pr`)
A SystemC process is a set ⟨`PMth`, `PTh`, `PCTh`⟩, where `PMth` is a method process (defined as a set ⟨`Mth`, `SC_SL`⟩ including the method and its sensitivity list), `PTh` is a thread process (accepts a wait statement in comparison to the method process), `PTh` is a clocked thread process (sensitive to the clock event).

Definition 2.5 (SystemC Program: `SC_Pg`)
A SystemC program is a set ⟨$L_{\texttt{SC\_Mod}}$, `SC_main`⟩, where $L_{\texttt{SC\_Mod}}$ is a set

of SystemC modules and SC_main is the main function in the program that performs the simulator initialization and contains the modules declarations.

2.2 Fixpoint Semantics

In this section, we define the semantics of the whole SystemC program, $\mathbb{W}$ ⟦SC_Pg⟧, and the SystemC module, $\mathbb{M}_{SC}$⟦m_sc⟧. Then, present the proofs (or proof sketches) of the soundness and completeness of $\mathbb{M}_{SC}$⟦m_sc⟧.

Definition 2.6 (Delta Delay: δ_d)
The SystemC simulator considers two phases *evaluate* and *update*. The separation between these two phases is called *delta delay*.

Definition 2.7 (SystemC Environment: SC_Env)
The SystemC environment is the summation of the default C++ environment (Env) as defined in [7] and the signal environment (Sig_Store) specific to SystemC: SC_Env = Env + Sig_Env = [Var → Addr]+ [SC_Sig → Addr,Addr], where Var is a set of variables, SC_Sig is a set of SystemC signals and Addr ⊆ ℕ is a set of addresses.

Definition 2.8 (SystemC Store: SC_Store)
The SystemC store is the summation of the default C++ store (Store) as defined in [7] and the signal store (Sig_Store): SC_Store = Store + Sig_Store = [Addr → Val]+ [(Addr, Addr) → (Val,Val)], where Val is a set of values such that SC_Env ⊆ Val.

Let $\mathtt{R}_0 \in \mathcal{P}(\mathtt{SC\_Env}\times\mathtt{SC\_Store})$ be a set of initial states, $\mathtt{pc}_{\mathtt{in}}$ be the entry point of the main function sc_main and $\rightarrow\subseteq$: (SC_Env × SC_Store) × (SC_Env×SC_Store) be a transition relation.

Definition 2.9 (Whole SystemC Program Semantics: $\mathbb{W}$ ⟦SC_Pg⟧)
Let $\mathtt{SC\_Pg} = \langle \mathtt{L}_{\mathtt{SC\_Mod}}, \mathtt{SC\_main}\rangle$ be a SystemC program. Then, the semantics of SC_Pg, $\mathbb{W}$ ⟦SC_Pg ⟧$\in \mathcal{P}(\mathtt{SC\_Env}\times\mathtt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathtt{SC\_Env}\times\ \mathtt{SC\_Store}))$ is

$$\mathbb{W}[\![\mathtt{SC\_Pg}]\!](\mathtt{R}_0) = \mathrm{lfp}^{\subseteq}_{\emptyset}\lambda X.\quad (\mathtt{R}_0) \cup \{\rho_0 \rightarrow\ \ldots\ \rho_n \rightarrow \rho_{n+1} |\ \rho_{n+1} \in (\mathtt{SC\_Env}\times \mathtt{SC\_Store}) \wedge \{\rho_0 \rightarrow\ \ldots\ \rho_n\} \in X \wedge \rho_n \rightarrow \rho_{n+1}\}$$

Both definitions of the semantics of process declaration ($\mathbb{P}_R$ ⟦SC_Pr⟧) and SystemC module constructor ($\mathbb{P}_{Ctr}$ ⟦SC_Ctr⟧) are given in [6]. In contrast to the semantics definition of an OO object in [7], a SystemC method can be activated either by the default context or by the SystemC simulator through the sensitivity list of the process. A complete definition of the semantics of a SystemC module object ($\mathbb{O}_{SC}$⟦o_sc⟧) through the definition of a transition

function $\texttt{next}_{\texttt{sc}}(\sigma)$=next($\sigma$) $\bigcup$ $\texttt{next}_{\texttt{sig}}(\sigma)$, including both parts C++ related and SystemC specific functions, can be found in [6].

Theorem 2.10 *Let*

$$F_{sc} = \lambda T.\quad \mathcal{S}_{\mathcal{O}}\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' | \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \textit{next}_{\texttt{sc}}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) = \cup_{n=0}^{\omega} {F_{sc}}^n(\emptyset)$

Proof. The proof is immediate from the fixpoint theorem in [1]. □

Definition 2.11 (SystemC Module Semantics: $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$))
Let $\texttt{m\_sc} = \langle \texttt{DMem}, \texttt{Ports}, \texttt{Chan}, \texttt{Mth}, \texttt{SC\_Ctr}\rangle$ be a SystemC module, then its semantics $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]) \in \mathcal{P}(\mathcal{T}(\Sigma))$ is:

$$\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!] = \{\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) \mid \texttt{o\_sc} \text{ is an instance of } \texttt{m\_sc},\ \texttt{v\_sc} \in \texttt{D\_in},\ \texttt{s\_sc} \in \texttt{SC\_Store}\}$$

Theorem 2.12 *(SystemC Module semantics in fixpoint) Let*

$$G_{sc}\langle S\rangle = \lambda T.\quad \{\mathcal{S}_{\mathcal{O}}\langle v, s\rangle \mid \langle v, s\rangle \in S\ \} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' | \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \textit{next}_{\texttt{sc}}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{M}_{SC}[\![\textit{m\_sc}]\!](v_{sc}, s_{sc}) = \textit{lfp}^{\subseteq}_{\emptyset}\ G_{sc}\langle\ D_{in} \times \textit{Store}\rangle$

Proof. Although the SystemC model presents some additional functionalities on top of C++, the proof of this theorem is similar to the proof of Theorem 3.2 in [7]. For instance, considering the definition of $\mathbb{M}_{SC}$ and applying in order Definition of a SystemC module object in [6], Theorem 2.10 and the fixpoint theorem in [1], the proof is straightforward. □

The last step in the SystemC fixpoint semantics is to relate the module semantics to the whole SystemC program semantics. Hence, we consider updated version of the function *abstract* (α°) as defined in [7]. The new function is upgraded to support the SystemC simulation semantics, environment and store. The complete definitions of $\alpha\_\texttt{SC}^{\circ}$ can be found in [6].

Theorem 2.13 *(Soundness of $\mathbb{M}_{SC}[\![\textit{m\_sc}]\!]$) Let M_{SC} be a whole SystemC program and let $m_{SC} \in M_{SC}$. Then*
$\forall\ R_0 \in \textit{SC\_Env} \times \textit{SC\_Store}.\ \forall\ \tau \in \mathcal{T}(\textit{SC\_Env} \times \textit{SC\_Store}).$

$$\tau \in \mathbb{W}[\![\textit{SC\_Pg}]\!](R_0) : \exists \tau' \in \mathbb{M}_{SC}[\![m_{SC}\ ]\!].\ \alpha\_\mathcal{SC}^{\circ}(\{\tau\}) = \{\tau'\}$$

Proof. (Sketch) We have to consider both cases when τ contains an object $\texttt{o}_{SC}$, instantiation of $\texttt{m}_{SC}$, and when it does not include any $\texttt{o}_{SC}$. For the second situation, the proof of the theorem is trivial considering that τ will be an empty trace. In the first case, the trace is not empty (let it be τ''). Since

SystemC modules are initialized in the main program `sc_main` before the simulation starts, there exist an initial environment, store and set of variables that define the initial trace $\sigma_0 \in \tau''$. The rest of the traces in τ'' are interaction states of $\mathtt{o}_{SC}$ because they are obtained by applying $\alpha\_\mathtt{SC}^{\circ}$ on τ. Therefore, $\tau'' \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}]\!]$. □

Theorem 2.14 *(Completeness of $\mathbb{M}_{SC}[\![]\!]$) Let $\mathtt{m}_{SC}$ be a SystemC module. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\quad \tau \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}\ ]\!]\colon \exists\ \mathtt{SC\_P} \in \langle L_{\mathtt{SC\_Pg}}\rangle.\ \exists \rho_0 \in \mathtt{SC\_Env}\times \mathtt{SC\_Store}.\ \exists\ \mathtt{o}_{SC}\ \textit{instance of}\ \mathtt{m}_{SC}.\ \textit{exists}\ \tau' \in \mathcal{T}(\mathtt{SC\_Env}\times \mathtt{SC\_Store}).\ \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge\ \alpha\_\mathtt{SC}^{\circ}(\{\tau'\}) = \{\tau\}$$

Proof. (Sketch) A SystemC program satisfying the previous theorem can be constructed by creating and instance of $\mathtt{m}_{SC}$ in the `sc_main` function, the initial state corresponds to the state when the module's constructor, `SC_Ctr`, was executed. An execution of a method of $\mathtt{m}_{SC}$ corresponds to executing a method thread (setting of the events in its sensitivity list to *Active*) and a change of a port corresponds to updating its internal signal by the new values. Hence, it is always possible to construct both `SC_P` and ρ_0. For instance, there exist many other possible constructions involving SystemC threads, clocked threads, etc. □

3 AsmL Fixpoint Semantics

AsmL [8] is one of the very latest languages developed for ASM [3]. It supports object-oriented modeling at higher level of abstraction in comparison to C++ or Java. We are going to restrict the AsmL semantics presented in this paper to the subset used in the program transformation.

3.1 Syntactical Domains

Definition 3.1 (AsmL Class: `AS_C`)
An AsmL class is a set $\langle$`AS_DMem`, `AS_Mth`, `AS_Ctr`$\rangle$, where `AS_DMem` is a set of the module data members, `AS_Mth` a set of methods (functions) definition and `AS_Ctr` is the module constructor.

One of the important features that we are going to use in AsmL corresponds to the methods pre-conditions (Boolean proposition verified before the execution of the method).

Definition 3.2 (AsmL Method: `AS_Mth`)
An AsmL method is a set $\langle$`AS_M`, `AS_Pre`, `AS_Pos`, `AS_Cst`$\rangle$, where `AS_M` is a the

core of the method, AS_Pre is a set of pre-conditions, AS_Pos is a set of post-conditions and AS_Cst is a set of constraints.

Note that AS_Pre, AS_Pos and AS_Cst share the same structure. They are differentiated in the methods by using a specific keyword for each of them (e.g., *require* for pre-conditions).

Definition 3.3 (AsmL Program: AS_Pg)
An AsmL Program is a set $\langle \mathtt{L_{AS\_C}}, \mathtt{INIT} \rangle$, where $\mathtt{L_{AS\_C}}$ is a set of AsmL classes and INIT is the main function in the program.

3.2 Fixpoint Semantics

Similar to the notion of delta delay (δ_d) of SystemC, AsmL considers two phases: *evaluate* and *update*. The program will be always running in the *evaluate* mode except if an update is requested. There are two types of updates, total and partial (usually performed using the Step instruction).

Definition 3.4 (AsmL Environment: AS_Env)
The AsmL Environment is a modified OO environment $\mathtt{AS\_Env} = [\mathtt{Var} \rightarrow \mathtt{Addr}, \mathtt{Addr}]$, where Var is a set of variables and $\mathtt{Addr} \subseteq \mathbb{N}$ is as set of addresses (two addresses store the current and new values of $\mathtt{v} \in \mathtt{Var}$).

Definition 3.5 (AsmL Store: AS_Store)
The AsmL store is $\mathtt{AS\_Store} = [(\mathtt{Addr}, \mathtt{Addr}) \rightarrow (\mathtt{Val}, \mathtt{Val})]$, where Val is a set of values such that $\mathtt{AS\_Env} \subseteq \mathtt{Val}$.

The whole AsmL program semantics ($\mathbb{W}_{AS}\ [\![\mathtt{AS\_Pg}]\!]$), method semantics ($\mathbb{M}_{AS}\ [\![.\ ]\!]$) and object semantics ($\mathbb{O}_{AS}[\![\mathtt{o\_AS}]\!]$) through the definition of a transition function $\mathtt{next_{as}}(\sigma)$ can be found in [5]. The AsmL class constructor is a can be defined according to the Definition 3.8 in [7].

Definition 3.6 (AsmL Class Semantics: $\mathbb{C}_{AS}[\![\mathtt{c\_as}]\!]$)
Let $\mathtt{c\_as} = \langle \mathtt{as\_dmem}, \mathtt{as\_mth}, \mathtt{as\_ctr} \rangle$ be an AsmL class, then its semantics $\mathbb{C}_{AS}[\![\mathtt{c\_as}]\!]) \in \mathcal{P}(\mathcal{T}(\Sigma))$ is:

$$\mathbb{C}_{as}[\![\mathtt{c\_as}]\!] = \{\mathbb{O}_{AS}[\![\mathtt{o\_as}]\!](v_{as}, s_{as}) \mid \mathtt{o\_as} \text{ is an instance of } \mathtt{c\_as},\ \mathtt{v\_as} \in \mathtt{D\_in},\ \mathtt{s\_as} \in \mathtt{SC\_Store}\}$$

Theorem 3.7 *(AsmL Class semantics in fixpoint) Let*

$$H_{as}\langle S \rangle = \lambda T.\ \{\mathbf{S_0}\langle v, s\rangle \mid \langle v, s\rangle \in S\ \} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \mathbf{next_{as}}(\sigma_n) \ni \langle \sigma', l' \rangle\}$$

Then $\mathbb{C}_{AS}[\![\mathtt{c\_as}]\!](v_{as}, s_{as}) = lfp^{\subseteq}_{\emptyset}\ H_{as}\langle\ D_{in} \times \mathbf{Store}\rangle$

Proof. Similar to the proof of Theorem 2.12 □

The function $\alpha\_\texttt{AS}^\circ$ is an updated version of the function *abstract* (α°) defined in [7]. The complete definition of $\alpha\_\texttt{AS}^\circ$ is given in [5].

Theorem 3.8 *(Soundness of $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!]$) Let P_{AS} be a whole SystemC program and let $c_{SC} \in C_{SC}$. Then*
$\forall\ R_0 \in \texttt{AS\_Env}\times\ \texttt{AS\_Store}.\ \forall\ \tau \in \mathcal{T}(\texttt{AS\_Env}\times\ \texttt{AS\_Store}).$
$\tau \in \mathbb{W}[\![\texttt{AS\_Pg}]\!](R_0) : \exists \tau' \in \mathbb{C}_{AS}[\![c_{AS}\ ]\!].\ \alpha\_\texttt{AS}^\circ(\{\tau\}) = \{\tau'\}$

Proof. Similar to Theorem 2.13. □

Theorem 3.9 *(Completeness of $\mathbb{C}_{AS}[\![]\!]$) Let c_{AS} be a SystemC module. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\quad \tau \in \mathbb{C}_{SC}[\![c_{SC}\ ]\!]\colon \exists\ \texttt{AS\_P} \in \langle L_{\texttt{AS\_Pg}}\rangle.\ \exists \rho_0 \in \texttt{AS\_Env}\times \texttt{AS\_Store}.\ \exists\ o_{AS}\ \textit{instance of}\ c_{AS}.\ \textit{exists}\ \tau' \in \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}).\ \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge\ \alpha\_\texttt{AS}^\circ(\{\tau'\}) = \{\tau\}$$

Proof. Similar construction approach to what we proposed in Theorem 2.14 except that instead of considering the SystemC method thread and sensitivity list, we consider here AsmL methods and their pre-conditions. Here also, there exist many possible constructions involving AsmL post-conditions, for example. □

4 Program Transformation

The equivalence in behavior, with respect to the observation α_o, between the source SystemC program and the target AsmL program is required to ensure the soundness of any verification result at the AsmL level. Our objective is to define a relation between the SystemC processes active for certain delta cycle and the set of methods allowed to be executed in the AsmL model. Hence, we will map every thread (method, sensitivity list) in the SystemC program by a method (method core, pre-condition) in the AsmL program to ensure having set of variables in both programs updated in the same time with the same values.

The SystemC observation function needs to see all the active processes at the beginning of a delta-cycle by checking for the end of the update phase.

Definition 4.1 (SystemC observation function: α_o^{SC})
Let $\texttt{SC\_Pg} = \langle \texttt{L}_{\texttt{SC\_Mod}}, \texttt{SC\_main}\rangle$ be a SystemC program, the observation function $\alpha_o^{SC} \in \mathcal{P}(\texttt{SC\_Env}\times\texttt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{SC\_Env}\times\ \texttt{SC\_Store}))$ is

$\alpha_o^{SC}[\![\texttt{SC\_Pg}]\!](\texttt{R}_0) =$

$$\text{lfp}^{\subseteq}_{\emptyset}\lambda X.(\mathtt{R}_0) \cup \{\tilde{\rho}_0 \rightarrow \ldots \tilde{\rho}_n |\ \forall \tilde{\rho}_i \in (\mathtt{SC\_Env} \times \mathtt{SC\_Store})\ \exists\ \{\rho^i_0 \rightarrow \ldots \rho^i_m\}$$
$$\in X \wedge \rho^i_m \rightarrow \tilde{\rho}_i \wedge \{\ \mathtt{m\_sc}\ \text{in}\ \mathbb{M}_{SC} \mid \exists \mathtt{o\_sc} \in \mathbb{M}_{SC}.\ \mathtt{o\_sc}(\rho^i_m()) \neq \{\epsilon\}\ \} = \emptyset\}$$

In the previous definition, α^{SC}_o is only tracing the initial states of a simulation cycle. For instance, the third condition confirms that in the last simulation cycle there was no single process ready to run. Similarly, we define an observation function α^{AS}_o for an AsmL program.

Definition 4.2 (AsmL observation function: α^{AS}_o)
Let $\mathtt{AS\_Pg} = \langle \mathtt{L_{AS\_C}}, \mathtt{INIT} \rangle$ be an AsmL program, the observation function $\alpha^{AS}_o \in \mathcal{P}(\mathtt{AS\_Env} \times \mathtt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store}))$ is

$$\alpha^{AS}_o[\![\mathtt{AS\_Pg}]\!](\mathtt{R}_0) =$$
$$\text{lfp}^{\subseteq}_{\emptyset}\lambda X.(\mathtt{R}_0) \cup \{\tilde{\rho}_0 \rightarrow \ldots\ \tilde{\rho}_n |\ \forall \tilde{\rho}_i \in (\mathtt{SC\_Env} \times \mathtt{AS\_Store})\ \exists\ \{\rho^i_0 \rightarrow\ \ldots\ \rho^i_m\}$$
$$\in X \wedge \rho^i_m \rightarrow \tilde{\rho}_i \wedge \{\ \mathtt{m\_as}\ \text{in}\ \mathbb{C}_{AS} \mid \exists \mathtt{o\_as} \in \mathbb{C}_{AS}.\ \mathtt{o\_as}(\rho^i_m()) \neq \{\epsilon\}\ \} = \emptyset\ \}$$

Next, we define the notion of equivalence between the two observations. Although, SystemC and AsmL have different environment and store structures, it is possible to ensure that they contain the same information.

Definition 4.3 (Equivalence w.r.t. α_o: $\equiv_{\alpha_o}$)
Let SC_Pg be a SystemC program, V_sc a set of its variables, AS_Pg be an AsmL program and Dout_as a set of its output variables.
prog_sc $\equiv_{\alpha_o}$ prog_as if

$\forall \mathtt{R}^{SC}_0$ set of initial states of SC_Pg. $\forall \mathtt{R}^{AS}_0$ set of initial states of AS_Pg.

$\forall \tilde{\rho} \in \{\tilde{\rho}_0 \rightarrow\ \ldots\ \rightarrow \tilde{\rho}_n\} \in \alpha^{SC}_o[\![\mathtt{SC\_Pg}]\!](\mathtt{R}^{SC}_0)$.

$\exists \hat{\rho} \in \{\hat{\rho}_0 \rightarrow\ \ldots\ \rightarrow \hat{\rho}_n\} \in \alpha^{AS}_o[\![\mathtt{AS\_Pg}]\!](\mathtt{R}^{AS}_0)$ |

$\forall$ vsc $\in$ V_sc. $\exists$ vas $\in$ V_as such that

if vsc $\in$ SC_Sig then $(\tilde{\rho}(\mathtt{vsc}) = (\mathtt{vl1},\mathtt{vl2})) \wedge (\hat{\rho}(\mathtt{vas}) = (\mathtt{vl1},\mathtt{vl2}))$

if vsc $\in$ AS_DMem then $(\tilde{\rho}(\mathtt{vsc}) = \mathtt{vl1}) \wedge (\hat{\rho}(\mathtt{vas}) = (\mathtt{vl1},\mathtt{vl1}))$

The observation function ensures that the AsmL program is mimicking the *evaluate* and *update* phases (same length n of the ρ sets). The first if condition takes care of the SystemC signals while the second one concerns basic C++ variables.

Theorem 4.4 *(Existence of transformed AsmL program w.r.t. α^{SC}_o) Let* ***SC_Pg*** *be a whole SystemC program,* ***SC_Din*** *a set of inputs and* ***SC_Dout*** *a set of out-*

puts. Then

$$\exists\ \texttt{AS\_Pg},\ \textit{an AsmL program, such that}\ \texttt{SC\_Pg} \equiv_{\alpha_o} \texttt{AS\_Pg}$$

Proof. (Sketch) The proof is done by constructing the AsmL program. For instance, for every SystemC module we affect an AsmL class having the same data members and methods. We set the pre-conditions, `AS_Ctr`, for the AsmL methods as a conjunction of the state of the events present in the sensitivity list, `SC_SL`, of the SystemC program processes. The tricky point in the construction is when to make the updates in the AsmL program. We have two possibilities: (1) C++ variables update: whenever a C++ variable is involved in an instruction, a partial update can be applied using the notion of *binders* in AsmL; and (2) SystemC signals: all signals are updated when all methods pre-conditions are false. Once the set of AsmL classes defined, Theorem 3.9 ensures the existent of the AsmL program. □

Theorem 4.5 *(Soundness of the transformation) Let* $\texttt{SC\_Pg}$ *be a whole SystemC program and let* $\texttt{AS\_Pg}$ *be a whole AsmL program. Then*

$$\texttt{SC\_Pg} \equiv_{\alpha_o} \texttt{AS\_Pg} : \quad \forall\ \texttt{Prop}(\texttt{V\_sc},\tilde{\rho}) \mid \tilde{\rho} \in \alpha_o^{SC}[\![\texttt{SC\_Pg}]\!].$$
$$\texttt{SC\_Pg} \vdash \texttt{Prop}(\texttt{V\_sc},\tilde{\rho}) : \texttt{AS\_Pg} \vdash \texttt{Prop}(\texttt{V\_as},\hat{\rho}) \mid \hat{\rho} \in \alpha_o^{AS}[\![\texttt{AS\_Pg}]\!].$$

where $\texttt{Prop}$ *is a program's property,* $\texttt{V\_sc}$ *is a set of variables of the SystemC program,* $\texttt{V\_as}$ *are their corresponding variables in the AsmL program.*

Proof. The proof is straightforward from the construction of equivalence relation $\equiv_{\alpha_o}$ in Definition 4.3. □

5 Conclusion

In this paper, we presented the fixpoint semantics of the SystemC library including, in particular, the semantics of a SystemC Module that we proved to be soundn and complete w.r.t. a trace semantics of a SystemC program. We provided also the semantics of a subset of AsmL and we proved the soundness and completeness of an AsmL class w.r.t. to a trace semantics of the AsmL program. Then, we proved the existence, for every SystemC program, of an AsmL program having similar behavior w.r.t. an observation function that we set to consider the traces of the system just after the update phase of the SystemC simulator. We have used this SystemC to AsmL transformation to reduce the complexity of SystemC models and enabled their formal verification [4] using model checking and theorem proving approaches used with AsmL and ASM languages in general.

References

[1] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, USA, 1979.

[2] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002.

[3] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[4] A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Design Automation and Test in Europe*, Munich, Germany.

[5] A. Habibi and S. Tahar. AsmL fixpoint semantics. Technical report, Department of ECE, Concordia University, December 2004.

[6] A. Habibi and S. Tahar. SystemC fixpoint semantics. Technical report, Department of ECE, Concordia University, December 2004.

[7] F. Logozzo. *Analyse Statique Modulaire de Langages à Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.

[8] Microsoft Corp. AsmL for Microsoft .NET framework, 2004.

[9] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Pub., 2003.

[10] P. D. Mosses. *Denotational semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science B.V., 1990.

[11] Open SystemC Initiative. Website: http://www.systemc.org, 2004.

[12] K. Oumalou, A. Habibi, and S. Tahar. Design for verification of a PCI bus in SystemC. In *Symposium on System-on-Chip*, Finland, November 2004.

[13] A. Salem. Formal semantics of synchronous SystemC. In *Design, Automation and Test in Europe Conference*, pages 376–381, Munich, Germany, 2003.